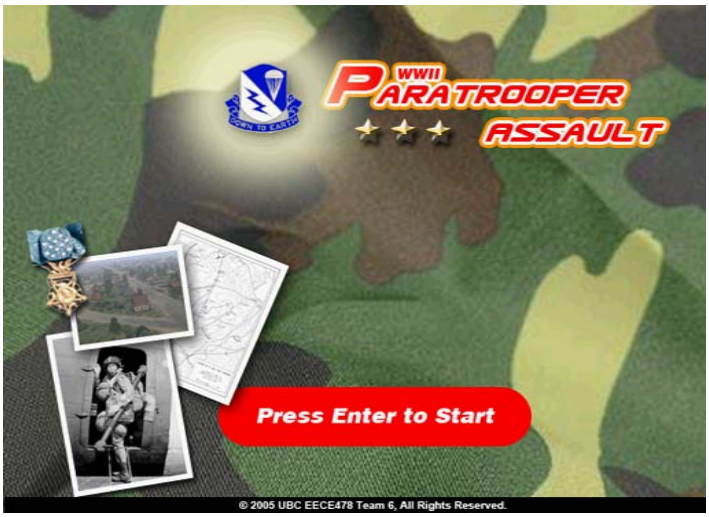
UNIVERSITY OF BRITISH COLUMBIA

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

EECE 478 –COMPUTER GRAPHICS

3D GAME DESIGN - WWII PARATROOPER ASSAULT

FINAL REPORT



TEAM 6:

Tzu-Heng Henry Lee Steven Shijia Ke

Yudi Sutanto Shane Wong

Max Yu

WINTER 2005

1 ABSTRACT

The aim of our game project is to design a 3-D computer game using OpenGL. The game we decided to create was a 3-D parachute game where the user can navigate a character through the sky and defeating obstacles along the way. The game was written in OpenGL using the SDL (Simple Direct Media Layer) library. As a result, we created a 3-D parachute game that contains all the essential components to make this game enjoyable and entertaining.

TABLE OF CONTENT

LIST OF FIGURES

2 INTRODUCTION

This report explains the approach to building a 3D computer game using the OpenGL language. By using the SDL libraries with OpenGL, we were able to create a 3D parachute game where the user parachutes through a series of waypoints and attempt to land near a moving tank to win the game. Due to the nature of a parachute game, and the fact that the environment the game is based in is very large, we used numerous game design techniques. Our main objective was to learn as much about the game creation process as we can. In the end, we were able to touch upon a huge area of game design, ranging from 3D model loading, to collision detection to game physic.

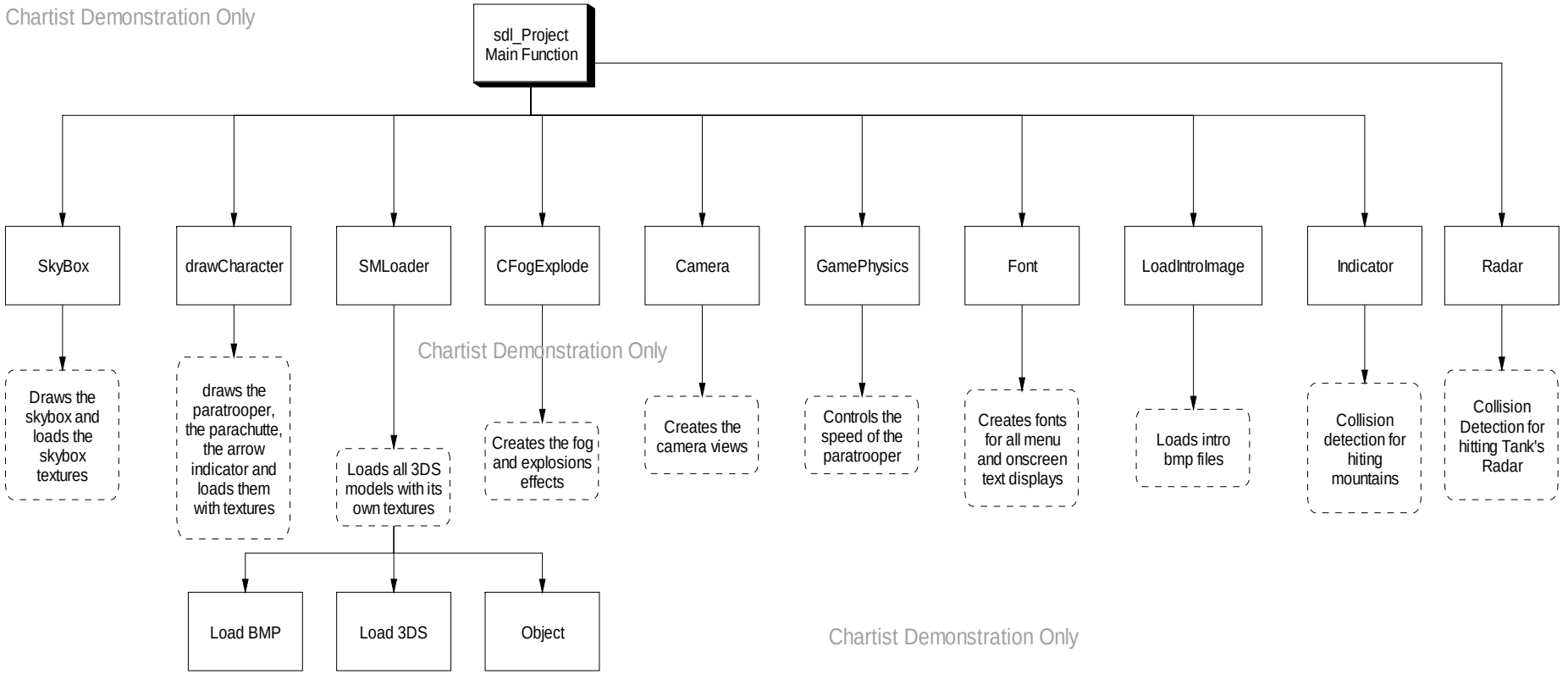
3 GAME DESCRIPTION

The WWII Paratrooper Assault game is based on WWII where the user plays the role of an Allied Paratrooper. The objective of this game is to land near a moving enemy tank to in order to capture it. Through this process, the user has to navigate through waypoints under constantly changing environment and dodge obstacles in order to reach the target. The user can control the paratrooper's movements by pressing the 'w', 's', 'a', and 'd' keys for up/down and left/right movements. In addition, the user can press the't' key to change between 3 different toggle view modes. Lastly, the user can hold down the left mouse button and drag the mouse in any direction to change the view.

4 IMPLMENTATION METHODS

The implementation of our game is broken down into several key sections. The first section shows an overview of the overall class structure. The next section goes into detail the methods used in each class structure to implement game functionalities.

4.1 CLASS FLOW DIAGRAM



4.2 IMPLEMENTATION DESCRIPTIONS

This section will list methods used to implement all the different aspects of the game.

4.2.1 3DS Loader and Animation

The loading of 3ds models contributed a huge part to the graphical interface of our game. Since we were short on time, we decided to load free 3ds models that are available from the internet to save time.

4.2.1.1 The 3ds Loader:

4.2.1.1.1 The First 2 Experimented Versions

There were 3 versions of the 3ds loader that we tried. Each one had its pros and cons. The first 2 3ds loaders were the L3DS loader from <http://www.levp.de/3d/3ds/>. That particular 3ds loader was extremely simple to use and consumed very little memory space. However, it could not texture the 3ds models and we were forced to find a different 3ds loader. The second 3ds loader we attempted to use was from <http://www.spacesimulator.net/tut4_3dsloader.html>. This particular loader allowed us to load an additional bitmap file that textures the 3ds model loaded. The limitation

with that class was it can only load one 3ds model at a time. This made loading 3ds models rather tedious and more complicated then necessary. Therefore, it was necessary to seek another alternative 3ds loader.

4.2.1.1.2 The Final 3ds Loader Version

The final 3ds version was taken from http://www.spacesimulator.net/tutorials.html (Tutorial #6). This version allowed us to load multiple 3ds models and texture them with individual bitmap files. The downside to this loader was that every transformation and rotation we make is cumulative. For example, if we translate a 3ds object to position (1, 1, 1) and then we want to move that to position (5, 1, 2). Due to the cumulative structure, we would need to move the object by (4, 0. 1). For linear transformations, this issue is rather simple to solve, however, when we tried to rotate the object, the axis would also be rotated. Therefore, if we want to translate that object after a rotation, we would need to find the new coordinates of the object's axis with respect to the global axis. One method we used to avoid this problem is to pre-translate all 3ds models in 3ds Studio Max before they are imported. The second method was to avoid rotation in the game and only rotate by 1 axis to make the math calculations easier.

4.2.1.2 Creating the 3ds Terrain

The 3ds terrain was custom made in 3d Studio by first creating a black and white bitmap picture shown below:

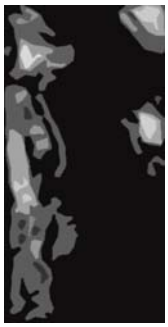


Figure 4.2 – Black and white bitmap picture

This picture is drawn using the brush tool in Photoshop then processed through with the Palette Knife effect. After which, we drew a plane in 3ds Studio and mapped the bitmap picture onto the plane using the translate effect to create the mountains from the contrast between the black and the white coolers. Finally we load this terrain into our game and texture-mapped it with a bitmap image and modified it according to our own specifications.

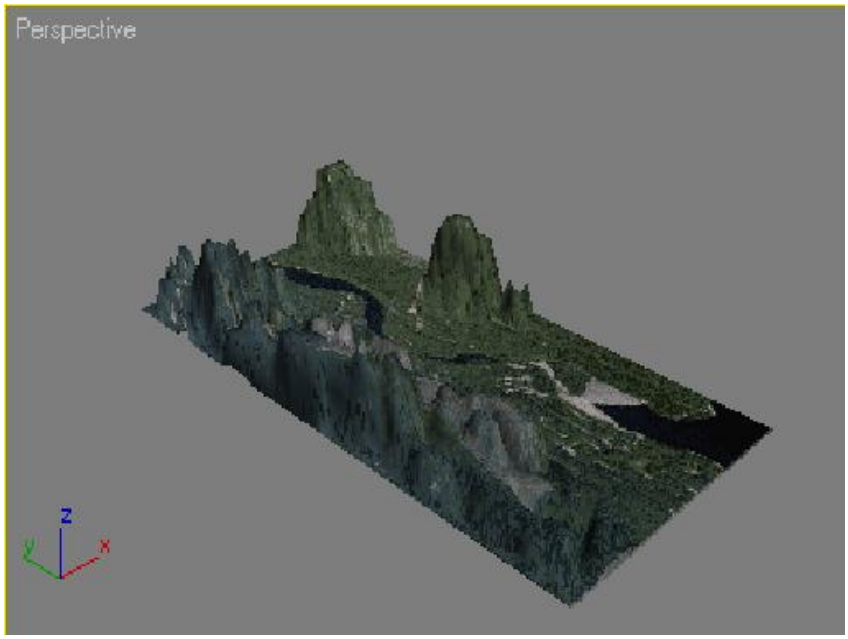


Figure 4.3 – Perspective view of the terrain

4.2.1.3 Loading Of 3ds Models

In total, we loaded a total of 9 different textures. They are listed and described in the table below.

Texture Name	Author	Description
Terrain	Own	Drawn in 3ds and Photoshop
Torus	Own	Used as waypoint in our game
MIG29	www.turbosquid.com	The first airplane fly by scene
Missile	www.turbosquid.com	Missile fired from the tank
Tank	www.turbosquid.com	Tank, our landing objective
Chapel	www.turbosquid.com	A structure for scenery beautification

Church	www.turbosquid.com	A structure for scenery beautification
Farm	www.turbosquid.com	A structure for scenery beautification
City	www.turbosquid.com	A structure for scenery beautification

These models are also accompanied by texture bitmap files that came with the 3ds model. The scaling and rotation of the models are processed in 3ds Studio Max. The positioning of these models in our game was done using trial and error method. We played the game extensively to find the best locations for these objects.

4.2.1.4 3ds Model Animation and the SMLoader Class

The animation of the 3ds objects were made through the SMLoader class we created. The SMLoader class basically loads all the 3ds models in its constructor and has various functions. The most important functions are the chooseModel and the CreateModel classes. The chooseModel function translates the 3ds object with the CreateModel acting as an initializer that creates and draws the 3ds model onto the screen in the beginning of execution.

To perform animation, the chooseModel function is called when certain conditions are satisfied. For example, if the character falls below a certain altitude level. By carefully keeping track of the translation and rotation magnitudes, we can create animation by simply redrawing the 3ds model at different locations with different orientations at

consecutive frames. The numbers of custom animations made were:

-MIG29 flies by

-Alien airplane disappears and reappears behind the character, then proceed to fly past the character.

-Tank movement. Currently we are using the pre-programmed method to move the tank in an assigned path way. Various artificial intelligence methods were tried. For example, we implemented a method where the tank was able to detect the relative position of the paratrooper and move in the opposite direction to avoid him. The challenging part of this method was that we can not keep track of where the tank is during game play. This created issues where the tank can run over buildings or into mountains. Since the collision detection method was only implemented on the paratrooper, we did not have time to find a method to apply that to the tank.

-Shooting of missiles from the tank towards the paratrooper. We also tried to use collision detection with this event. The difficulty lied in the methods we used to move the paratrooper. Since the paratrooper is falling based on a timer of the computer's CPU clock speed, the exact falling speed of the paratrooper varies from computer to computer. Therefore, when we were calculating the position of the missiles and the paratrooper, we were not getting the exact location of the paratrooper. Hence, our collision detection was acting very strangely. Therefore,

we did not use that method during our final game version.

4.2.2 Collision Detection

Since this game a simulation of the reality, the real physical environment. We need to model the game as realistic as it can. An important factor that makes the game realistic is the collision detection. For example, one cannot run and pass through a concrete wall (without getting hurt—if you must argue). Our concept of collision detection for our game has two aspects. The first aspect is the detection of the intrusion of an object into a specific space. The second aspect is the response of this intrusion.

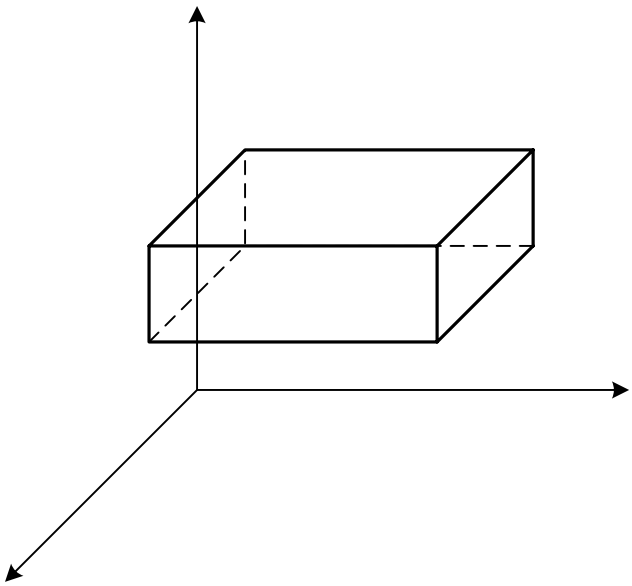


Figure 4.4 – Space definition

In implementing the detection of an object intruding a specific space, we followed a series of steps. We first defined a specific space with several coordinates. This is shown is figure 4.4 those coordinates resemble a virtual box. We found that defining a box is more intuitive than defining anything else. Thus, in our parachuting game, we adapted box as our general tool to describe spaces that the user can or cannot enter. To calculate if an object (for example, the parachuter) is within the boundary of this space or not, we could take the present coordinate of that object and compare it with the description of the space. Below is a sample script that demonstrate this process.

```
if( userPosition.x > box.x1 && userPosition.x < box.x2
   && userPosition.y > box.y1 && userPosition.y < box.y2
   && userPosition.z > box.z1 && userPosition.z < box.z2 )
{
   gameWorld.response();
}
```

It is important to have fresh data of an object's current position in doing calculation of collision detection. In our program, we tried to align our code so that the collision

detection routine would be called not long after the new object position had been queried.

The response of the intrusion of an object into a space can vary according to different scenarios of the game. In our game there are three types of response of collision detection. The first type of response is to do nothing and let the object maintain its physical property (position, velocity, acceleration). In our game we defined the spaces that were close to the mountain as “danger zones.” If a user drifted into one of these spaces, his velocity will maintain the same, since he is still in the air.

The second response is semi-elastic collision. According to Newton’s Law, when an object makes contact with another object, it exerts forces on the other object, but at the same time it also receives forces from the other object. In our game we had attempted to make the character jump when he lands on a surface. However, we were only able to make the character stop (velocity set to zero) when he touches the surface.

The third response, in an interesting way, can also be referred as “no response.” In this type of collision detection, your game state changes in response to the collision. For example, you would be directed to the “game-over” scene if you touch the mountain anytime in the game. You would also be directed to the “mission accomplished” scene when you maneuver the character to land on the tank at the end of the game.

It should be mentioned that this type of collision response is actually not really a collision detection technique; however, it is often used to replace scenes that need a more sophisticated level of collision detection calculation that the code and the system requirement cannot compromise. For example, it would be hard to implement how the person's result velocity is when he land on a rugged surface. He might fall and tilt on one side if the surface normal is not completely vertical, and it would also be hard to take all the vertices into our account in collision detection because there were more than one thousand vertexes for the landscape we were using in our game. It was also hard to determine how elastic the collision is between the paratrooper and other surfaces. The ultimate example we have in the game was when the paratrooper lands on the tank. It was impossible to estimate the response without conducting an in-depth research. To avoid chores like these, we considered it feasible to just avoid the response. The result was actually satisfactory.

4.2.3 Game Physics

In our game we need a game physics class that helps us to decide how our character will be moving in the air. Our approach in implementing the game physics involves two aspects. The first aspect is the environmental factor that causes change in the game. The second aspect is the calculation of the resulting movement of our character.

The environmental factors that can inflict change on the character movement can be many. In making the game physics class of our game it was hard in the beginning to decide how complicated the game physics class can be. Due to the time constraint in working on this project, we decided to make it as simple as possible. In our game we took account of the gravitational force, the turbulence of the air, and the gesture of the person as factors that may contribute to the change in his movement in the air. Canonically speaking, to describe a character's movement in the air we would have parameters that define the position, velocity, and acceleration with the Gaussian coordinates. Among these we can change the acceleration and velocity only. Below is a summary of these factors and their influence to the velocity and acceleration of that character in the game.

Factor	**Influence**
Gravitational force	Downward acceleration. Used to calculate existing velocity.[1]
Exisitng velocity	Present velocity. Used to calculate the present position
Wind	Change the present horizontal and vertical velocity. Used to calculate the present position.

Gesture (character tilting forward)	Forward velocity increases.[2]
Gesture (character tilting backward)	Backward velocity. [2]
Gesture (character tilting to the right)	Velocity to the right side. [2]
Gesture (character tilting to the left)	Velocity to the left side. [2]

[1] Since the character will be parachuting, we will not see much effect of acceleration (other than the fact that the gravitational acceleration do contribute to pull the user down and not up).

[2] The horizontal velocity increases in this way, because if the parachute is not upright the air under the parachute will push it toward the side it is tilted.

The second aspect of our game physics class is just the calculation of the present acceleration, velocity, and most importantly the position. The formulae we used to implement this calculation is the all-so-familiar integration of the instantaneous velocity and velocity. These formulae are listed below.

$$v_{present} = a_{present} \times \partial t$$

$$d_{present} = \frac{1}{2} a_{present} \times (\partial t)^2 + v_{present} \times \partial t + d_{previous}$$

Since our program would be calculating and updating the new position of the character at an incredible speed (thanks to the CPU and graphic card), we assumed

that we can get a differential time value that is very small and therefore is able to give us a reasonable value. However, the challenge we had in our implementation of the calculation was that we were not able to use the <time.h> or <ctime.h> to get a time value that's less than one second. After doing some research we found that we can actually use the CPU's clock cycle to get small time value. In C++ there are two helper function that determines the number of ticks a CPU can have in one second and how many ticks it has so far. These two are QueryPerformanceFrequency() and QueryPerformanceCounter(). A simple script for calculating a differential time value is shown below.

```
LARGE_INTEGER ticksPerSecond;

LARGE_INTEGER tick;   // A point in time


// get the high resolution counter's accuracy

QueryPerformanceFrequency(&ticksPerSecond);

double CPUFrequency = ticksPerSecond.QuadPart;
```

```
// what time is it?

QueryPerformanceCounter(&tick);

double NumOfTicks1 = tick.QuadPart;



//********* after some time…..*******//



// what time is it now?

QueryPerformanceCounter(&tick);

double NumOfTicks2 = tick.QuadPart;



// calculating the time difference:

double diffTime = (NumOfTicks2 – NumOfTicks1)/ CPUFrequency;
```

4.2.4 Menu System

The menu system is build using a while function inside while function. The main menu system is enabled through setting two variables called main_menu_is_on and

menu_is_running to true.

If the main menu is executed, it will clear the buffer screen and depth buffer, and then menu choices will be outputted. Before, clearing the buffer screen, one need to save the View Matrix, and then pop back the View Matrix when the user has done configuring the settings.

The game menu has four choices:

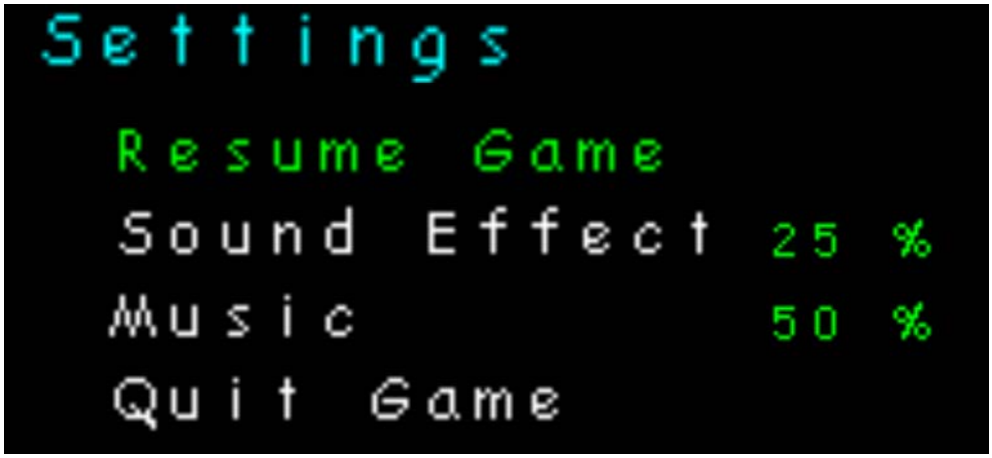


Figure 4.5 – The game menu

The sound effect and the music's volume can be adjusted. The default is as shown above. When a user adjusts the volume, he or she will hear harp sound. This will indicate the desired volume not just the number that should up on the screen.

4.2.5 Sounds

The sound implemented in the PARACHUTING GAME is implemented using SDL_MIXER library. This library divides sound into two categories which are music and sound effect. The music is the sound played on the background and it is not event driven; thus, it always playing when a user plays the game. The sound effect, however, is an event driven, it will be executed when a user starts press certain key. In addition, the sound effect will also be turned on when a user is at certain altitude. For instance if he is 1400 m above the ground, the sound of wind will be played, and when a user is 350 m above the ground, the sound of tank is played.

The SDL_MIXER has a feature on the sound effect which enables multiple sound effects played at the same time. The multiple sound effects are enabled through the utilization of channels, for instance, one can mix the sound of a tank on channel 1 and missile on channel 2. The programmer has to be efficient on assigning the channels, because there are fewer channels available then the sound effects. This means that each sound effect is not assigned on a channel. A programmer has to be efficient.

In the implementation for our game, there are four channels being used. The first channel, channel 0, is used for the most important sound such as character movement, toggle view and menu screen. Channel 1, 2 and 3 are used for less important sound. Channel 1 is used for sound effect like wind, tank and wave point. The last two channels are being utilized for some additional sound like voices that we recorded.

The problem that we encounter with the SDL_MIXER is setting the volume for the background music. We, however, have no problem with the volume setting on the sound effect. The only problem arises when set a wave file (.wav) as the background music. Apparently, if we use that specific file, the volume setting is not implemented correctly on the SDL_MIXER API (extern DECLSPEC int SDLCALL Mix_VolumeMusic(int volume). Thus, we cannot solve this problem unless we change the library. To encounter the problem, we just replace the background music with different format (.xm, .it). The volume setting functions well in these types. There are many background music available from the web, one example is from modarchive(http://www.modarchive.com/artists/acoustic/).

Because we can solve the volume setting by replacing wave file with .xm file, we enable to set the volume. As a result, we have an additional feature for our game which is volume setting. Another additional feature is as we get closer to the tank the sound of the tank becomes louder.

4.2.6 Environment Effects

4.2.6.1 Explosions Effect and the CFogExplode Class

The explosions effect was based on a tutorial at NeHe, tutorial #19. The code was modified slightly to create the effect of a particle exploding with adjusted number of

exploding particles and particle texture. The limitations of this effect are the explosions can only be defined on a 2-D plane. This limited our ability to blow certain objects up because the user can rotate and adjust the view, therefore we will not be able to track the position of explosion from a 3-D coordinate system into a 2-D system.

4.2.6.2 Fog Effect and the CFogExplode Class

The fog effect was based on lesson #16 at NeHe.com. The code was modified to integrate properly into our game. We also created different fog levels to show a more apparent environment effect on the game. The wind effect was added to complement the fog effect. The wind effect can alter the paratrooper's x, y, and z velocities randomly. This is achieved by using a random number generator. The difficult part of this was to test and make sure the random number generated was within a reasonable range to make the game difficult yet beatable.

4.2.7 Pre-Programmed Events

The checkEvent function is responsible for storing all the events that are pre-programmed to occur. These events are usually set off depending on the altitude of our character. We used the altitude because the rate the character falls can not be controlled by the user. These events are written by assigning state variables and

increment to the next state when the present actions have been completed. When a certain event is set off, different codes are executed depending on the event. For example, an event can set off codes to translate a 3ds model or set off explosions onto the screen.

4.2.8 Way Points and In Game Indicators

4.2.8.1 Way Point detection

The way point detection is accomplished by constantly checking the parachute troopers' position. If the distance between the character y position and way point y position is less than 3, and the distance between the character and the way point in x-z plane is less than 20, a unique piece of music will be played to notify the user that he/she has successfully pass through the waypoint. On the other hand, if the player missed the way point, his or her health will be decreased by 1 unit.

4.2.8.2 Direction Indicator (Arrow)

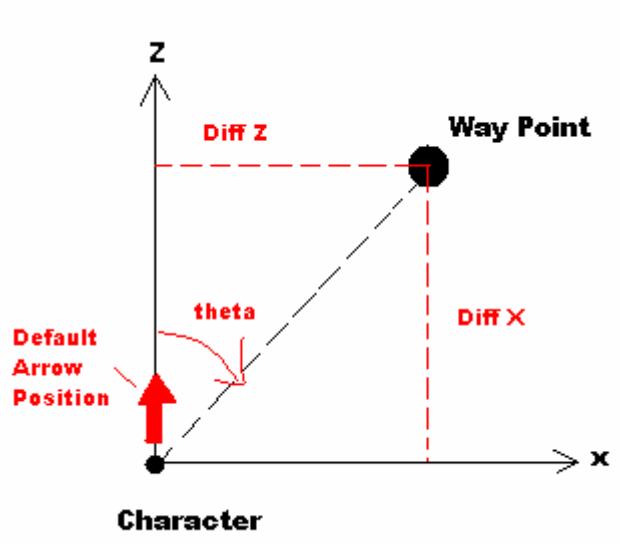


Figure 4.6 - Arrow Rotation.

An arrow is drawn in drawCharacter.cpp using a pyramid and 4 small cubes. The blinking effect of the arrow is achieved by changing the color of each individual component of the arrow once at a time depending on how many times the code is executed. To point the arrow to a way point, difference in distance z and difference in distance x as shown in figure 4.6 are calculated. Therefore the angle theta can be found by using the following equation:

$$\theta = \frac{180.0}{\pi} \times \tan^{-1}(\frac{\text{Xdiff}}{\text{Zdiff}})$$

This angle theta is then passed to setCharacter function in drawCharacter.cpp where the arrow is drawn and rotated. Theta will determine the angle of rotation of the arrow.

4.2.8.3 Danger Zone Indicator

The use of indicators increases the interactivity the user can have with the game and therefore making the game more fun and attractive. A class (Indicator.cpp) is written for this game to increase the interactivity with the user by signaling the user that he/she is approaching or is currently in a danger zone that will cause the user to lost life points. Please refer to other section (e.g. collision detection, game pausing) for a deeper discussion of this class.

4.2.9 Camera View

4.2.9.1 Camera Class

This class was constructed based on the concept of spherical coordinates.

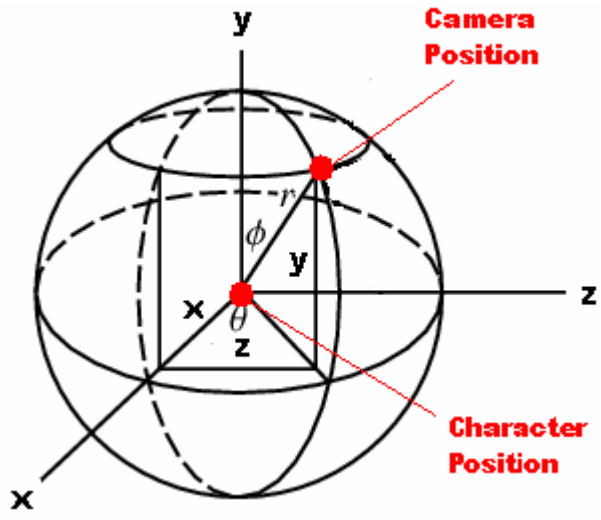


Figure 4.7 – Character Relations in Spherical Coordinates

Figure 4.6 demonstrates the relationship between the camera and the game character. Angle ϕ represents the angle between r and the y-axis while angle θ represents the inverse tangent of z / x. The constructer of this class simply sets up the default camera position. The position is calculated by the following equations:

$$\text{eyex\_cam} = \text{r} \times sin(\theta) \times sin(\phi) + \text{Body\_Position\_x}$$

$$\text{eyey\_cam} = \text{r} \times cos(\phi) + \text{Body\_Position\_y}$$

$$\text{eyez\_cam} = \text{r} \times cos(\theta) \times sin(\phi) + \text{Body\_Position\_z}$$

In those equations, the addition of Body_Position_x, Body_Position_y, and Body_Position_z at the end allows the camera to move with the character. The camera always looks into the center of the character, and that is Body_Position_x,

Body_Position_y, and Body_Position_z.

The function SetView( ) sets up the camera model matrix and employs gluLookAt function which takes in eyex_cam,eyey_cam,eyez_cam, atx_cam,aty_cam,atz_cam, upx_cam, and upy_cam,upz_cam as parameters.

Toggle( ) function allows user to switch different toggle views. This function is called when user presses “t”. (see User Inputs). Users can also rotate the camera to a desired position by using the mouse.

ChangePos function takes in the current character position and updates the new camera position by recalculating the x, y, z position for camera. This is necessary because the position of the character changes continuously.

4.2.9.1 Functions for Rotating the Camera(Using Mouse-Drag Events)

Phi_inc(),Phi_dec(),Theta_inc(), and Theta_dec() functions are called in the SDL_Project.cpp when a combination of SDL mouse events take place. Mouse-Drag event takes place when left button of the mouse is clicked, mouse motion is detected, and the direction of the mouse motion is detected. For example, if the left mouse button is pressed and the mouse is moved to right, Theta_inc() will be called. (see User Inputs for Mouse-Drag Instructions)

4.2.10 Game Pausing

Time Pausing in Danger Zone Indicator Class (Indicator.cpp):

In creating a function for pausing the danger zone indicator of our game we needed to utilize three things. The first one is the Boolean value that records if the game was paused at that point of time. The second one is the timer that gives you the present time. The third one is a variable that let you store the time information you need to keep track of. A simple flow chart that demonstrates the pausing process of the indicator class is shown below.

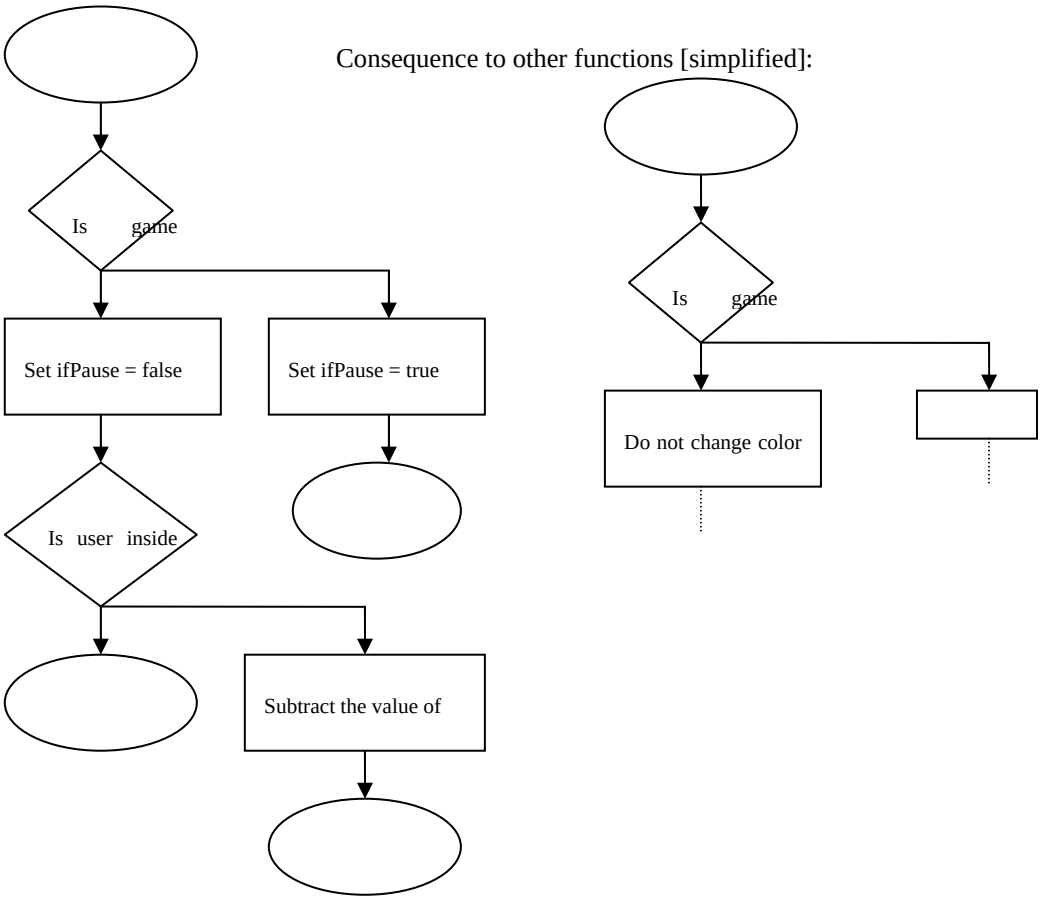


Figure 4.8 – Flow chart of the pausing process of the indicator class

When the pause function is called, we first check if the game has been already been pause (by checking the ifPause Boolean value). If the game has not been paused, we would set the ifPause to true. At the same time, we would also record the present time value of the game to keep record. The game state of the indicator would stop from this point. If the game has been previously paused, this function will then resume the game by setting the ifPause value to false. The present time value will also be stored in the variable resumeAt. This function will then examine if the user is inside the danger zone box or not. If user is within the danger zone, it possible that he/she has been staying inside for an amount of time (indicated by the total staying time in the

flow chart). We would subtract the amount of paused game time from the total staying time in order to get the actual, accurate staying time. The game state of the indicator would resume from this point.

4.2.11 Drawing the Paratrooper

The character model was first designed in GLUT using simple objects such as spheres and cylinders to represent different body parts. Out character model consisted of head, shoulders, upper arms, arm joints, lower arms, hands, thighs knees, legs, heels and feet. The parachute was drawn using the Bezier surface method. A control point grid of 4x4 was used. We calculated these points based on the 4 corners of the parachute defined by the 4 lines we drew coming out of the backpack of the paratrooper. In order to make the implementation of the character movement easier, the hierarchical model method is applied. By using the method, we were able to express the relationships between parts of the character model. In this case, the character's body is the parent node. The diagram below shows the relationship between the parent node and its child nodes and terminal nodes:

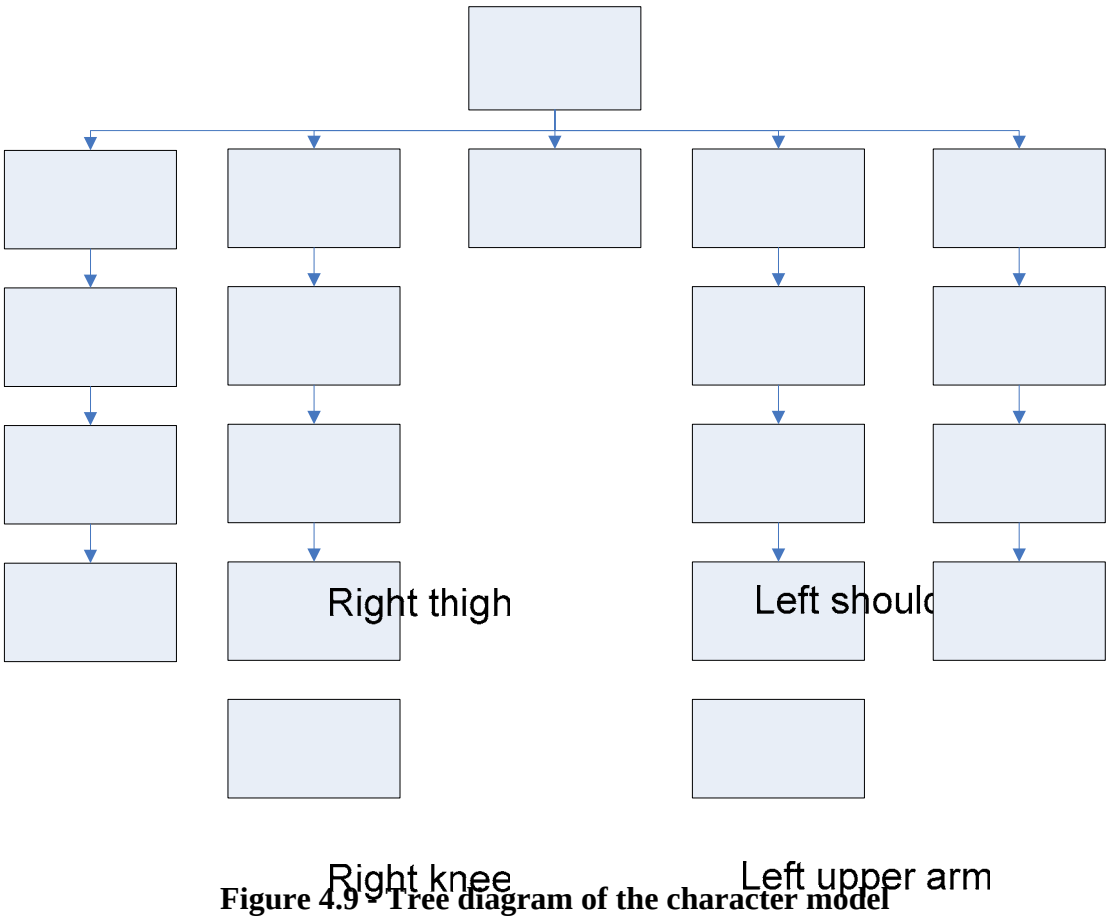


Figure 4.9 - Tree diagram of the character model

In our character model, some components are identical such as right shoulder and left shoulder, right upper arm and left upper arm. For objects that are identical, a single prototype was created. As a result of this information storing method, the program became more efficient. Depending on the keyboard pressed by the user, the character will react differently such as leaning forward, backward, right or left. As mentioned above, the character model movement is designed by utilizing the hierarchical model method. Therefore, depending on which key is pressed, the joint angles between components will increase or decrease with respect to the components to which they are attached. For instance, when the user pressed the forward key "w", the character's body will lean forward. To explain further, the body will first rotate around the Y-axis

and so will the other components since they are all body's child nodes. If the user pressed the forward key again, the angle rotate around the Y-axis increases and the character model will tilt even more towards the forward direction. On the other hand, if the right/left key is pressed, the character model will move toward the right/left and different joint angles for different components will rotate around different axis to simulate a real human movement. Next, if the user first wants the character to lean forward and then move right, the character, the character will first lean back to its initial position and then move back.

The next task is to texture map the character. We chose to use paint to draw texture pictures for the character model because it will be saved in BMP format which is more detailed than any other picture file format. Moreover, it is easier to work with since there are a lot of internet sources for loading this format into OpenGL. After saving all the pixels in the picture, we saved it into the texture memory. Then, the texture is bound and it is ready to be loaded into OpenGL. And from OpenGL, different textures are mapped onto different objects.

4.2.12 MPEG Loading

Loading an MPEG video as our introduction video of our game was accomplished by using the SMPG library. In addition, we also used a tutorial from http://www.utdallas.edu/~dbb033000/SMPEG/Playing%20MPEG%20files%20with%

20SMPEG.htm as a reference to load the mpg files into memory. We simply declared a set of SDL initiation variables for the intro video to be displayed and then re-declared another set of SDL initialization variables to start our game.

4.2.13 Font

The font implemented in this game uses the bitmapped font. The idea was taken from Nehe Tutorial Lesson 15 (http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=15). The figure below illustrates the bitmapped font. The code on the Nehe tutorial was not complete because it only works in black background. In many games, the background is not black, thus we need to modify the code. We will have the black background if we did not modify the code. To do the modification, we blend the bitmapped font so that the background can be removed. The other added future for the font is enlarging the font.

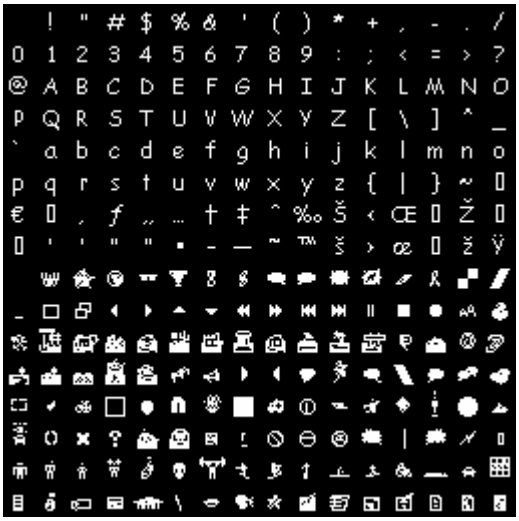


Figure 4.10 - Bitmapped Font

The font is used to inform a user about the altitude, health, level and time. The altitude is between 2400 and 0 m. The maximum health is 4 and there are two levels in this game.

The time that we currently have on the screen was ideally used for limiting the time to complete the game. However, after several discussions that we have had during the meeting, we decided to make the game more fun by disabling the time limit.

To create a font, the first thing to do is to search for bitmapped pictures. The bitmapped font is then texture mapped and sliced. Each sliced would define an alphabet and they will be stored in a list. Once the function is called, it will automatically display the list.

4.2.14 Skybox

Skybox is a cube surrounding the parachute which is used to manipulate a sky view for a user. The skybox acts an infinite world for the user. If the skybox had not been implemented then a user would have felt that the world in our game was finite.

The skybox is always following the user and the camera. The size of the skybox is approximately 25 by 25 unit for each cube. The figure of the skybox can be shown here.

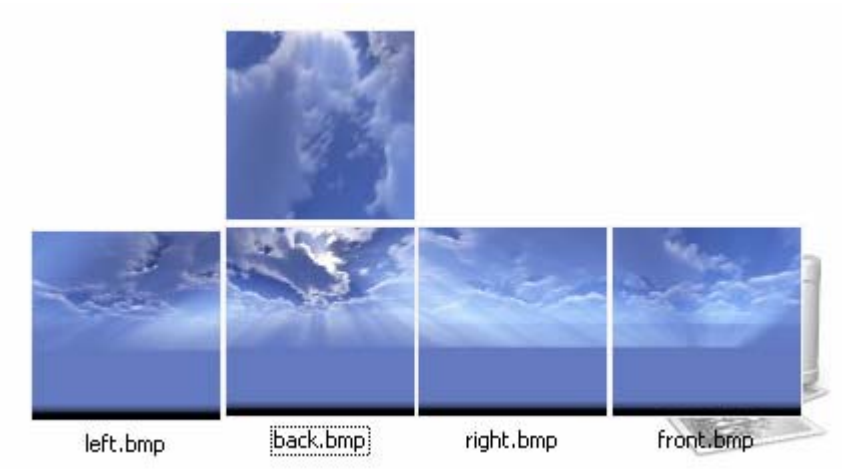


Figure 4.11 - The Skybox pictures

5 POSSBILE IMPROVEMENTS

Due to time constraints the paratrooper game can undergo several future improvements to enhance the game. This section lists possible game improvements we would recommend.

5.1 GRAPHICS

To extract an algorithm for the character's movement proved to be difficult during the process of producing this game. However, if more time was allowed, the character's movement can be improved. First of all, we can improve the character movement so that whenever the character is moving from an axis to a different axis, the character will gradually return to its initial position and then to the axis that the user want it to be moved. As shown in the figure below:

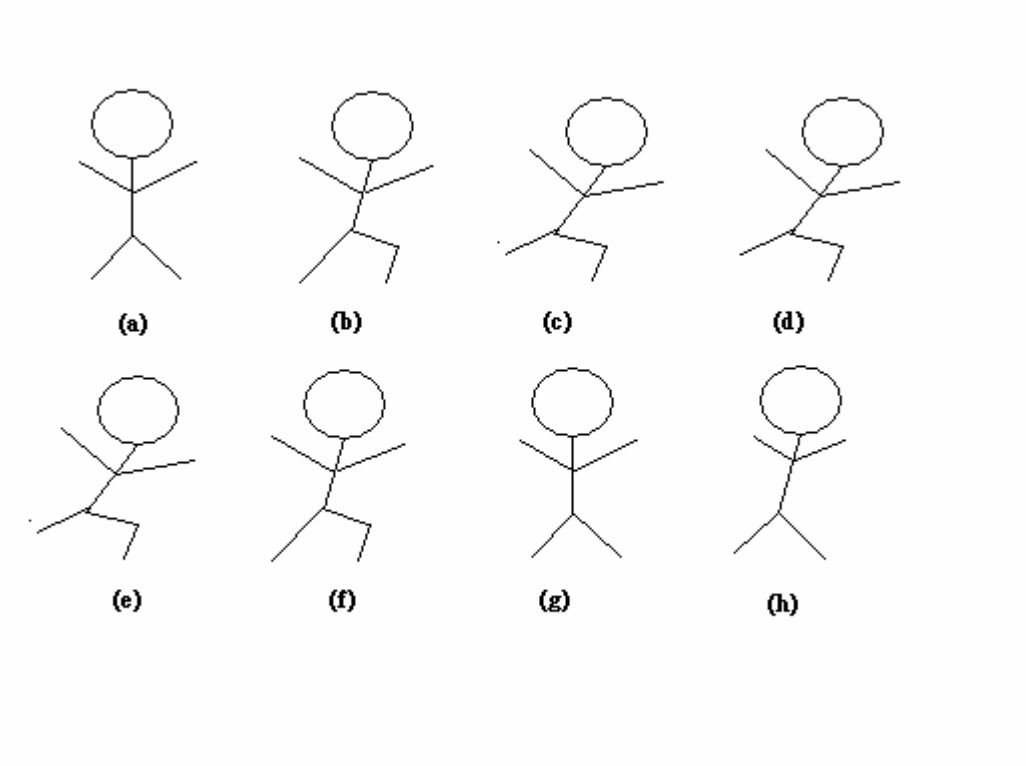


Figure 5.1 – Character movement

In (a), the character is in its initial position and the user is about to press the “right” key. In (b)-(d), we can see that as more “right” keys are pressed, the character’s body angle tilts more towards the right. Then, the user presses the “forward” key. As we can see from (e)-(g), the character gradually returns to its initial position. Finally in (h), the character starts to move forward.

Another part of the character model and animation we can improve on is to make the character model looks more realistic. Due to the limited open source for texture pictures for OpenGL on the internet, we were forced to draw the character’s body parts using Paint program. As a result, the character model does not look realistic. Although we tried to find programs that will help us produce better visual effects for

the character such as, they are proven to be incompatible for OpenGL. Another feature we can add to the character model is the facial expression such as eyes blinking. Moreover, the character can be designed so that it can express emotions such as happy, anger, sadness, and nervous.

5.2 GAME PLAY

The game would probably be better if the game play is little bit faster but has nice terrain. A user would feel stunned if the view is decent so he will enjoy browsing through the game. A wonderful terrain can be found when playing Grand Theft Auto: San Andreas. There will be a time when the main character will be asked to jump with a parachute from a building. The scene after that is stunning. If we could use the city view it would be really great.

A moving wave point should probably add some more aspects to game playability; it makes the game much harder, and thus frustrates the user and keeps him playing. Improvements regarding game physics would be to add more effects such as wind, moving plane, etc to the game.

6 CONCLUSION

In conclusion, the parachute game we produced has all the important features such as environment visual effect, sound effect and collision detection to make the game feasible and playable. Needless to say, if given more time, a lot of the features can be enhanced to make the game more interesting and attractive. First of all, the visual effects such as the game environment, the parachute trooper and the moving objects can be dramatically improved. In addition, the game play can be designed to be more challenging. This can be accomplished by adding moving way points, and moving objects that will crash into the parachute trooper.

7 APPENDIX

The executable and all source codes of our game are posted on our eece478 wiki Team 6 website. The individual progress logs are also posted there as well.

8 REFERENCES

User inputs

Camera rotates upward	[mouse] left click the mouse without release and move the mouse up
Camera rotates downward	[mouse] left click the mouse without release and move the mouse down
Camera rotates to right	[mouse] left click the mouse without release and move the mouse right
Camera rotates to left	[mouse] left click the mouse without release and move the mouse left
Switch Toggle Mode	[keyboard] press “t”

Source of reference

1. Lev Povalahev. 3D graphics using the OpenGL API. 1999-2002. 12 Nov. 2005. <http://www.levp.de/3d/>

2. Damiano Vitulli. Spacesimulator.net. 2000-2005. 16 Nov. 2005. <http://www.spacesimulator.net/>

3. Turbo Squid. 2005. Turbo Squid, Inc. 29 Oct. 2005. <www.turbosquid.com>

4. The Mod Archive. 1996-2005. Modarchive.com. 22 Nov. 2005. <http://www.modarchive.com/artists/acoustic/>

5. Drew Benton. (2005). *Playing MPEG files with SMPEG*. Retrieved November 23, 2005 from World wide Web:

<http://www.utdallas.edu/~dbb033000/SMPEG/Playing%20MPEG%20files%20with%20SMPEG.htm>

6. NeHe. 2005. Neon Helium. 25 Nov. 2005. <http://nehe.gamedev.net/>